Final report

Contract No. F61775-99-WE072

22nd May 2001

Authors:
Prof. P. Nixon, Dr. S Dobson and P. Barron

Contact details:
Department of Computer Science
Trinity College Dublin
Ireland
Email: Simon.Dobson@cs.tcd.ie

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From – To)*
29-04-2002	Final Report	20 August 1999 - 20-Aug-00

4. TITLE AND SUBTITLE

Dynamic Reconfiguration Of FPGA Nodes In A Distributed Computing System: A Preliminary Investigation

5a. CONTRACT NUMBER
F61775-99-WE072

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Professor Patrick Authur Nixon, Dr. S Dobson, P. Barron

5d. PROJECT NUMBER

5d. TASK NUMBER

5e. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Trinity College, Dublin
Dublin
Ireland

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

EOARD
PSC 802 BOX 14
FPO 09499-0014

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
SPC 99-4072

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This report results from a contract tasking Trinity College, Dublin to investigate a specialized portion of a heterogeneous information system, specifically, Field Programmable Gate Array (FPGA)-based nodes. New computing architectures will be investigated that can specify and dedicate FPGA processing elements (FPE) that take advantage of application/algorithm dependent dataflow. This research will be presented in three interim reports, describing the integration of a FPGA system and a Corba control architecture as detailed in the technical proposal. As well, a final specification, design and evaluation report will be delivered.

15. SUBJECT TERMS
EOARD, Distributed Computing

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18, NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Christopher Reuter, Ph. D.
UNCLAS	UNCLAS	UNCLAS	UL	35	19b. TELEPHONE NUMBER *(Include area code)* +44 (0)20 7514 4474

(1) In accordance with Defense Federal Acquisition Regulation 252.227-7036, Declaration of Technical Data Conformity (Jan 1997), All technical data delivered under this contract shall be accompanied by the following written declaration:

"The Contractor, Trinity College, Dublin, hereby declares that, to the best of its knowledge and belief, the technical data delivered herewith under Contract No. F61775-99-WE072 is complete, accurate, and complies with all requirements of the contract."

DATE: _____

Name and Title of Authorized Official:

_____
(End of Clause)

(2) In accordance with the requirements in Federal Acquisition Regulation 52.227-13, Patent Rights-Acquisition by the U.S. Government (Jun 1989), CONTRACTOR WILL INCLUDE IN THE FINAL REPORT ONE OF THE FOLLOWING STATEMENTS:

~~(A) "Disclosures of all subject inventions as defined in FAR 52.227-13 have been reported in accordance with this clause."~~

~~Or,~~

(B) "I certify that there were no subject inventions to declare as defined in FAR 52.227-13, during the performance of this contract."

DATE: _____

Name and Title of Authorized Official: _____

The final stage of this project initially was to focus on FPGA integration of the software architecture developed in deliverable 2. However, the key expertise in FPGA development for the project left the University. Therefore, in the spirit of the original proposal we have pursued a slightly different line of investigation to complete the work. We believe this is consistent with the original aims of the proposal and provides a strong platform to follow through the FPGA investigation at a later stage.

1 Introduction

An architecture [deliverable 2] has been developed to aid in the mobility of objects throughout their environment and to further support fault tolerant rebinding. The next phase is the creation of a system that will assist in the communication between these mobile objects and the building itself. To this purpose a distributed event service is to be developed.

The aim of this document is to present a design for a distributed event service that will complement the architecture in [1]. An example of such a service would be the policy service. The purpose of this service is to assign the rights and the wishes of the mobile objects within the object's environment.

Section two discusses some of the requirements that are seen to be important in the design of this particular event service. Section three presents a design.

2 Requirements

As this event service will be part of a smart building environment, in which mobile objects will play a central part of the infrastructure. It will be necessary that the requirements for such a system take these factors into consideration. At this point in the design stage the requirements are as follows:

?? The ability of the event service to scale at a reasonable rate. One of the problems, which the event service will have to overcome, is event storming[1].
?? To support mobile objects. As an example, if a mobile object wishes to move nodes. The event notifications that it received at the current node have to be forwarded onto the object's destination.
?? Support the function of filters/constrains within the event service. To help in the reduction of unwanted events. A filter/constrain will only notify an object of the particular events that they are interested in.
?? Support the policy service. The policy server will hold the rights and the wishes for each mobile object. It is therefor necessary for the event service to interact with this service to insure that the right measures are being taken.
?? The event service is also required to insure the event notifications are delivered at least once to each object.
?? Also the event service needs to delivery the events in the order that they occurred[2].

[1] Event Storming is the uncontrolled notification of events to clients, who may or may not wish to receive the notification of events.
[2] While this requirement is desirable it is not total necessary at this stage of the project.

3 Proposed Design

These sections will hopefully proposed an architecture that will fulfil the requirements stated in the section 2 of this document. Some of the concepts that were used in the building of previous event services will be used and put into practice in the design of the event service for the smart building.

3.1 Overview of Design

Before entering into the details of the design, it would be helpful to clarify some of the terms that will used to explain the design. A *device* will be considered as any object that able to execute pieces of code and has access to the network. Whether it is connected to a wireless network or a more conventional LAN. Such a device could be a PC, laptop or even PalmPilot. *Mobile objects* are equivalent to the Active Objects (AO) in the JEDI [2][3] event model. Which are defined as autonomous computational units performing application-specific tasks.

In general there are two basic types of events services. One, which makes the producer of an event, distributed the event to all the interested parties. This type is known as the Push Model. The second is where the object with interest in an event polls the producer for new events. And this is known as the Pull Model. For this design the Push Model will be implemented. There is also another defining characteristic of event services. It is in the way which clients receive there notification of an event. Corba Event Service [5] uses the *event channel* approach, where clients attach to the *event channel* to gain access to new events. The other approach that is used, and which will be implemented in this design, is where clients subscribe to an event. This can be seen in the JEDI [2][3] event model or in the Cambridge Event Model [4]. So in summary the event service that is described in the sections below will be based on clients subscribing to events. And where clients will receive notification of the events, using the Push Model as described above.

3.2 Main Components

The main components of this architecture are the devices and mobile objects that occupy it. These devices support mobility of mobile objects from one device to another. The architecture for the mobility of these objects is covered under [1]. However these devices now need to support mobile objects in the producing and the consuming of the events. To this aim an *event bus* will support this. The *event bus* is not unlike the Event Dispatcher described in the JEDI [2][3] event model. In this model an *event bus* will be located on each device. Its role is to support mobile objects in the location of events, subscription to events, notification of events and the delivery of the events. Also, its role is to support mobile objects in moving location. Insuring that the mobile objects still receives the notifications of the events that it has subscribed to.

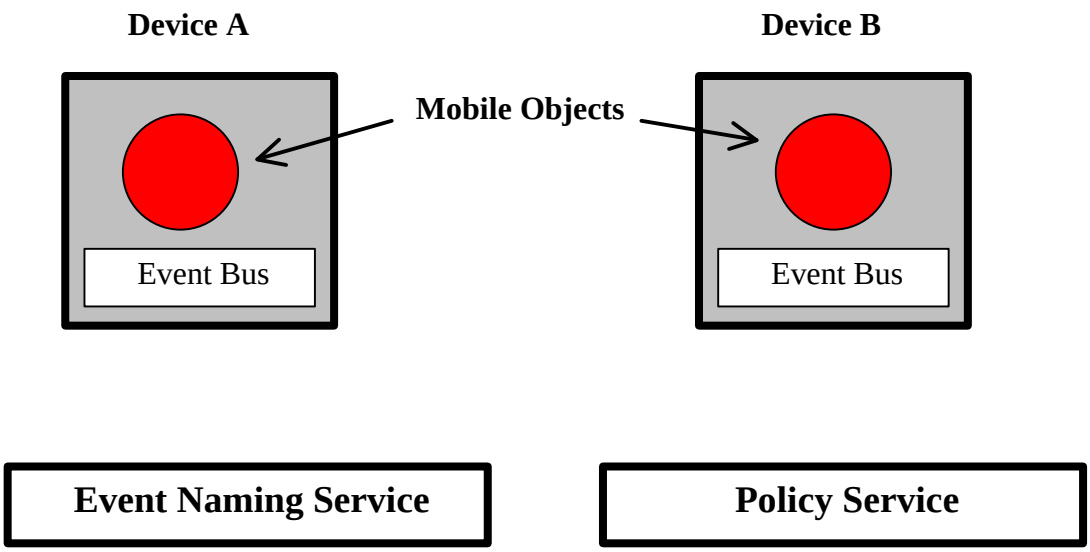


Figure 1 Main Components

To enable the event buses to locate the source of an event, a naming service is used to query the whereabouts of the event bus that is supporting that event. This allows the separation of the actual name of the event and the location to where the event source is. Which is a benefit when the actual location of the source of the event might change due to the movement of the event source (mobile object) from one device to another.

It is also possible to conceive that there would be more than one location for a particular event. To support this the Event Naming Service would need to keep multiple entries for each event. And on request for the location of an event it would supply the whereabouts of all the source of that event.

The purpose of the Policy service is to hold the rights and the wishes for each mobile object within the smart building environment. The event service uses this service to create filters for the notification of events to the mobile objects. However the design of the Policy service is outside the scope of this document.

3.3 Naming of Events

In order for interested parties to find and show interest in the different events that occur within the system, a naming system must be adopted whereby the events can be uniquely identify throughout the system. For this a URI type scheme will be used for the naming of the events. Note that with this architecture the name of the event only locates the server that holds the information on the event. Such as the location of the event bus that are supporting the source of the event.

An example URI that could uniquely identify an event might look like the following, *EDP::://wilde.cs.tcd.ie/Oreilly/floor2/paddy/enterevent*. This would uniquely identify the event, but also give the location to where the information on the event is to do found and what protocol is to be used.

3.4 Advertising of Events

In Siena [6] sources of events have first to advertise their intention to publish events to the event service. The same approach will be used in this event service to indicate the readiness of the event source to produce events. But, also to show the location of the source.

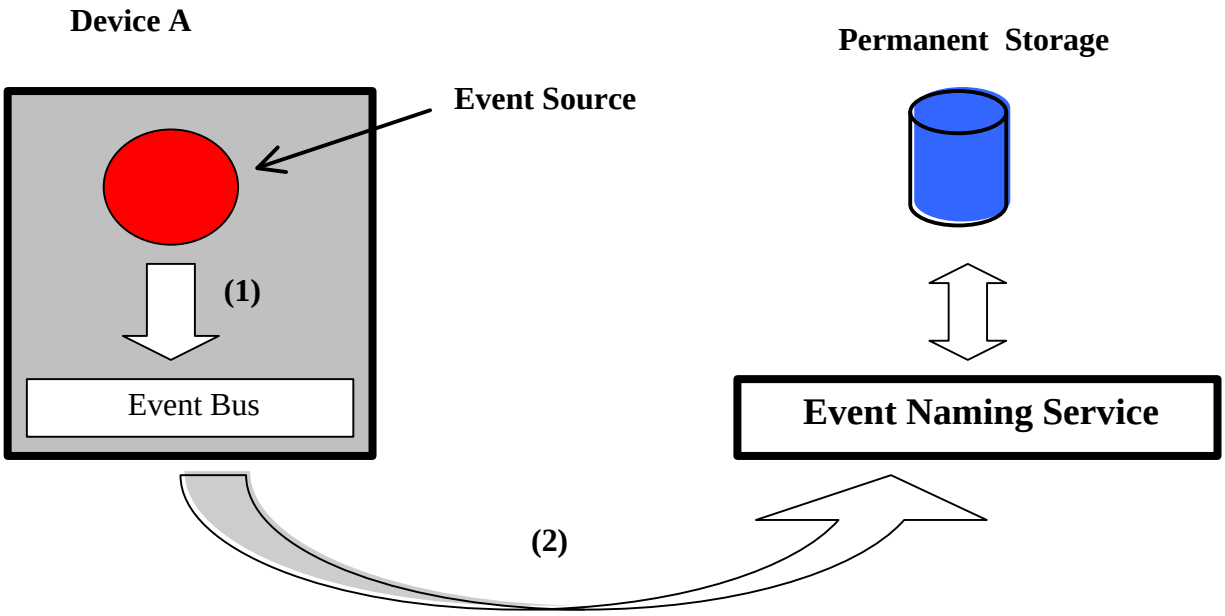


Figure 2 Advertisement of an event

The first step in the advertising an event is the event source, which in most case will be a mobile object, call *Advertise(name, other info)* method to indicate to the event bus it readiness to produce events. Advertise function pass the event bus the name of the event, which is the identifier discussed in section 4.3. It also passes other information such as owner event source.

The next step is for the local event bus to forward this information, along with location of the event source, onto the Event Naming Service specified in the name of the event. After the Naming Service receives this information for the new event, it inserts the information into its database or adds the event source to the list of other sources that are creating that particular event. Once this is completed the event source is considered as been advertised its event.

3.5 Subscribing to Events

As with most event services they require the clients in some way to subscribe to events that they are interested in. This narrows the scope of the events on the overall system. It also prevents event storming by only sending the notification of the event to the parties, which have shown an interest in that event.

Clients in this event service are required to subscribe to an event using the full name of the event, as described in section 4.3 of this document. It is also possible for the

client to attach a filter[3] when subscribing to an event. Thought it is not necessary as the event bus in conjunction with the policy server can define one.

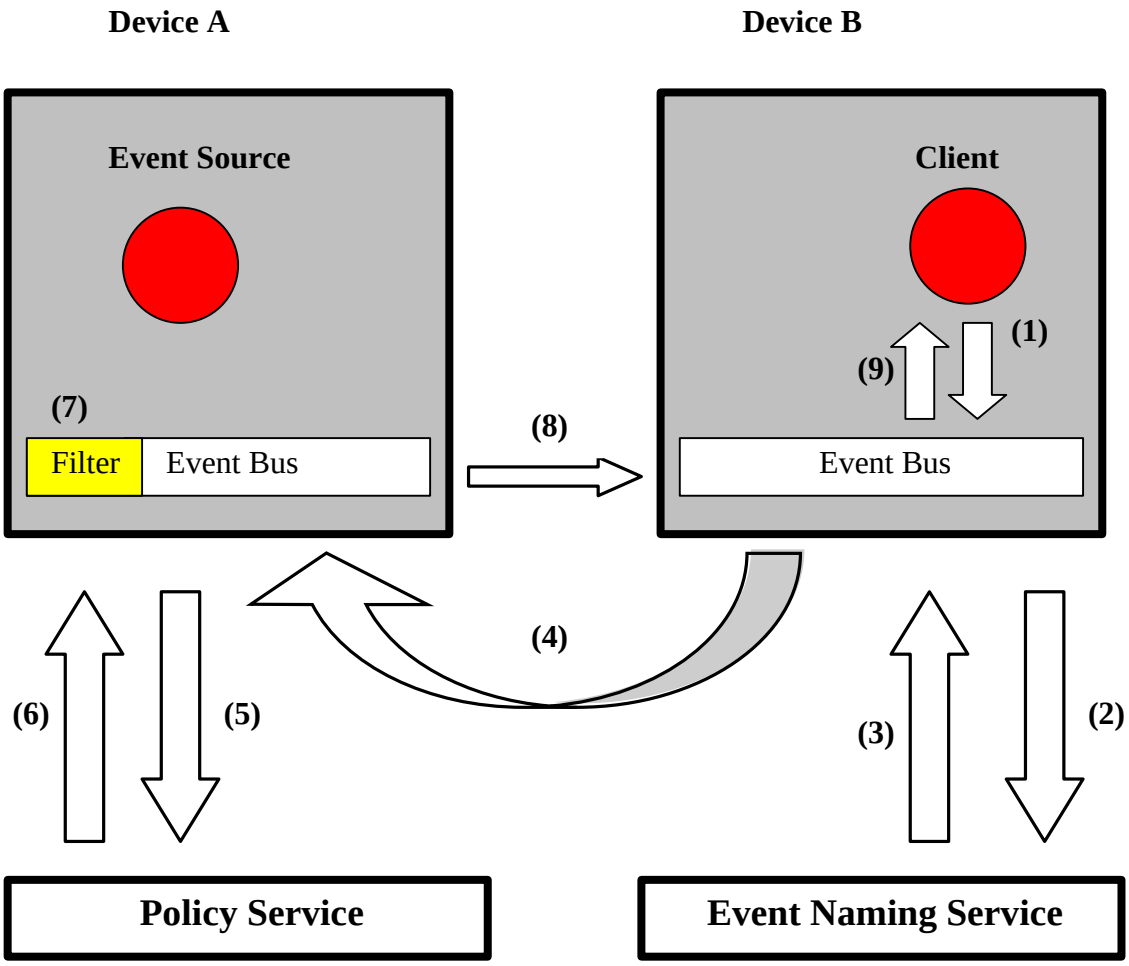


Figure 3 Client subscribing to events

Figure 3 shows the series of steps required for a client to subscribe to an event. There are four distinct stages to subscribing; the client making the request, finding the location of the event bus(es) that are supporting the event source, subscribing to the event bus(es) and the installation of the filter.

In the first stage (figure 3 step 1) the client requests its local event bus to subscribe it to a certain event. It dose this using a *subscribe(name, other info, filter)* method. This method passes to the event bus the name of the event that the client is interested in. Also other information that would identify the mobile object, which is needed by the policy server when obtaining information for the filter. A user defined filter can also specified. Once clients event bus has received the request from the client, it must locate where the sources of the event are, which is accomplished with the help of the Event Naming Service (figure 3 steps 2,3). On receiving the locations of the sources,

[3] A filter is a means by which the client receiving notification of event can specify which events it wishes to be notified about in more detail (see section 3.8)

the event bus subscribes to each one, passing information about the mobile object making the request, the name of the event and the user defined filter if specified (figure3 step 4).

When the event bus supporting the event source receives a subscribe request, it must first install the filter associated with the mobile object and add it to list parties that are interested in that particular event. There are two ways in which a filter can be obtained. Either by using the filter that was defined by the mobile object or by requesting information from the policy server (figure 3 steps 5,6) to construct the filter. On completion the client mobile object is notified of the success or failure of the operation (figure 3 step 8,9).

3.6 Unsubscribing from Events

Unsubscribing operation is the opposite of subscribing, which was defined in the previous section. It is executed when mobile objects no longer wish to receive notification of an event.

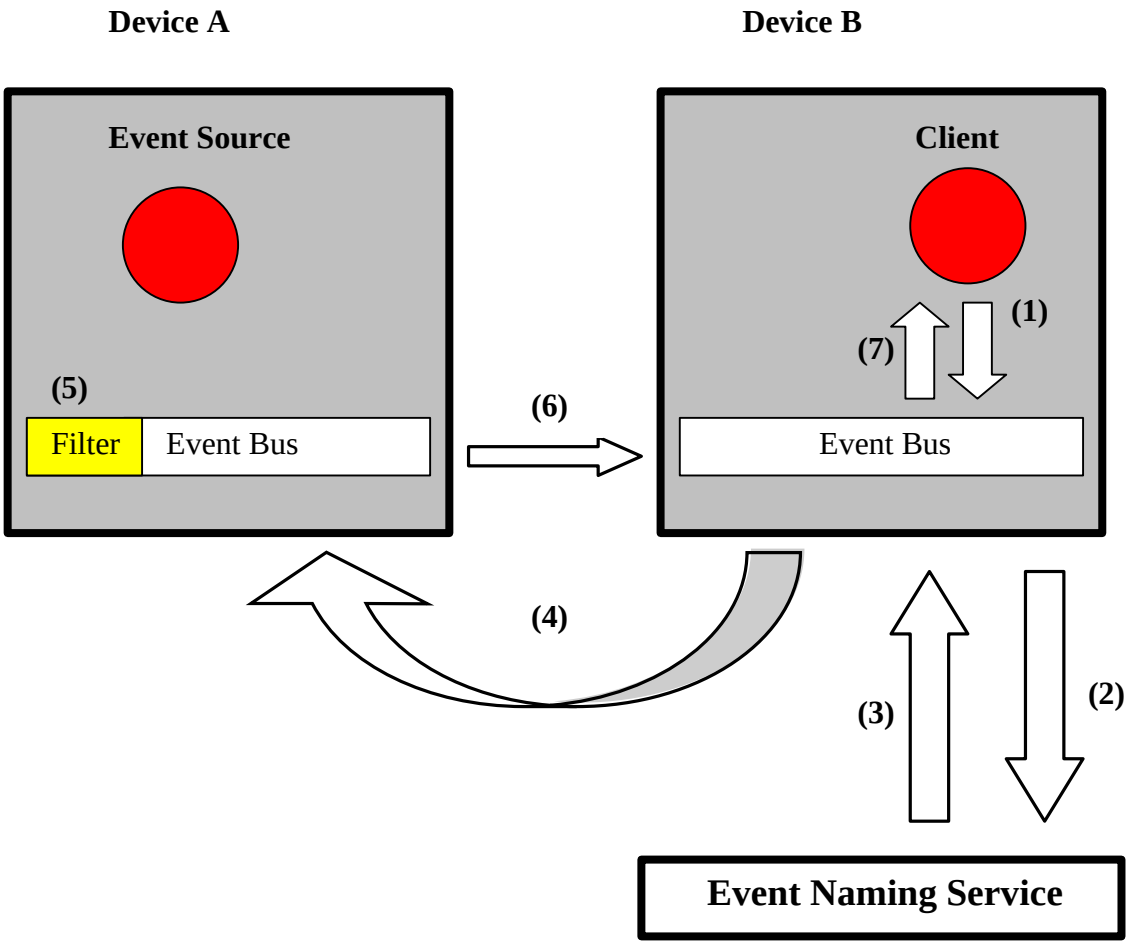


Figure 4 Mobile object Unsubscribing from an event

To ensure that the event bus has the right location of the event sources when unsubscribing from an event, it obtains the location of the event source from the Event Naming Service (figure 4 steps 2,3). This may not be necessary in most cases as the event sources would not have changed since subscribing to the event, but to

support the mobility of the event source a lookup is necessary to ensure the right location is obtained.

3.7 Notification of Events

Once events have been triggered by an event source it is necessary that the interested parties be notified of the event. It is the aim of this architecture that the interested parties receive this notification at lease once and receive the events in the order that they occurred at the source.

To overcome these problems each event is to be numbered in the order that they have occur at the event source. When the client's event bus receives notification of the event, it places them in the right order as they occurred on at the source. The event bus only delivery's the events to the client (mobile object) in the right sequence. If for some reason an event is missing, the client's event bus can query the event source's event bus for the missing event.

However the above will only guarantee that the events from a particular event source are in the right order. If there were multiple locations of the event them this architecture could not guarantee the order in which they would arrived in.

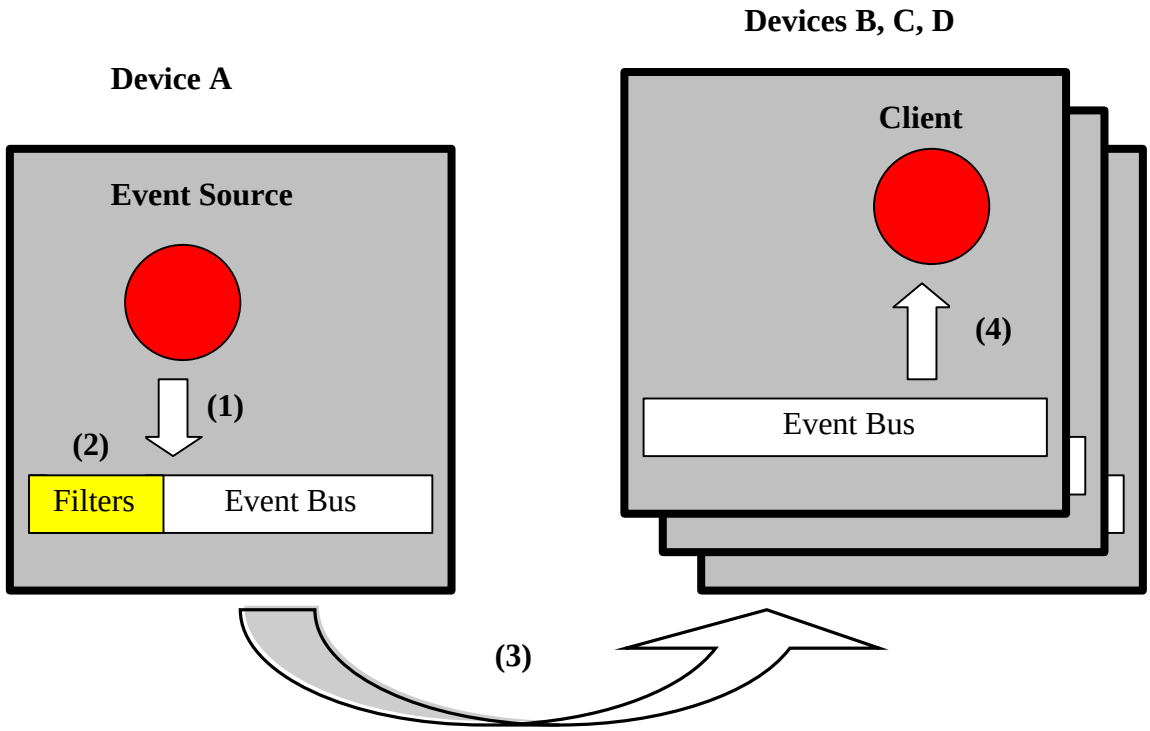


Figure 5 Notification of an event to interest parties

In normal scheme of things a typical notification might possibly look like that shown in figure 5. The event source notifies it local event bus of a new event and passes the parameters associated with this instance of the event. The job of the event bus is to then to notify the interested parties of the event. It however only notifies the mobile objects that have subscribed to the event and have passed the filter supplied by the mobile object or created from the policy server. The client's event bus at appropriate time, delivery's the event to the client (mobile object).

3.8 Retracing Advertisements

When source of events stopped executing and has finished producing events. It is requirement that event bus supporting the source retracts the advertisement made at he beginning of the event's life. The process quite similar to that of the *advertising*, but in his case the event bus requests the Event Naming Service to delete the source's entry instead of adding or inserting it.

3.9 Filters

Filters are a concept used by many event services to stop events that the consumer is not interested in receiving. Both Siena [6] and COBEA [7] use this concept of filters to help in the notifying interested parties of events occurring.

Within this architecture a mobile object can specify the filters, it is also possible to be created by the policy service in conjunction with the event bus. In this case, as in [7], the filter is placed at the source of the event to decrease the network traffic to consumers. An example of such a filter could specify that mail client only wishes to receive mail for Simon that is less than 10k in size.

3.10 Support for Mobile Objects

As the event service is going to be located within a mobile environment it is necessary that service is able to support mobile objects in their movement through the environment. This requires that when a mobile object wishes to move devices any events that the object has subscribed to are changed, so that the notification of events are forwarded onto the new location. If however the mobile object is a source for events, all the information associated with mobile object must be passed onto the new location of the object.

The only event service that comes close to supporting mobile objects is JEDI [2][3]. It defines a particular type *active object* [2], which they call a *reactive object.* This object has the ability to support mobility. When this object decides to migrate to another device, the following occurs:

?? The state of the reactive object is serialised and saved using standard Java facilities.
?? The reactive object moves to the new location and informs the Event Dispatcher that it is ready to receive events.
?? The Event Dispatcher keeps the events that should be received by the migration reactive object until it is ready to receive them.

It is proposed to do something similar in supporting mobility of objects within this event service. However this service most able to support mobile objects which produce events and ones which consume events. This implies that two separate approaches have to be taken in dealing with each case. In the case of a mobile object which consumes events the mobile object is required to carry out the following steps (also see figure 6):

1. The mobile object notifies it local event bus of its intention to move to a different location.
2. The local event bus first finds the location of all the events sources by querying the Event Naming Service. The local bus informs the event buses, which support the event sources, of the intention of the mobile object to move devices. Each

source's event bus is required to buffer all the events that occur in the relocation of the mobile object.

3. The mobile object is then able to move to its new location.
4. On arriving at new device, it signals to the local event bus that it is mobile object wishing to redirect notifications to this location.
5. The queries the Event Naming Service for the location of the event.
6. Event Naming Service supplies the event bus with the locations of the event sources.
7. The local event bus forwards a request onto each source's event bus of the new location of the mobile object and sets the necessary objects to receive notification.
8. Each source's event bus updates their subscription list (SL) and notifies the mobile object of any events that it has missed while relocating.

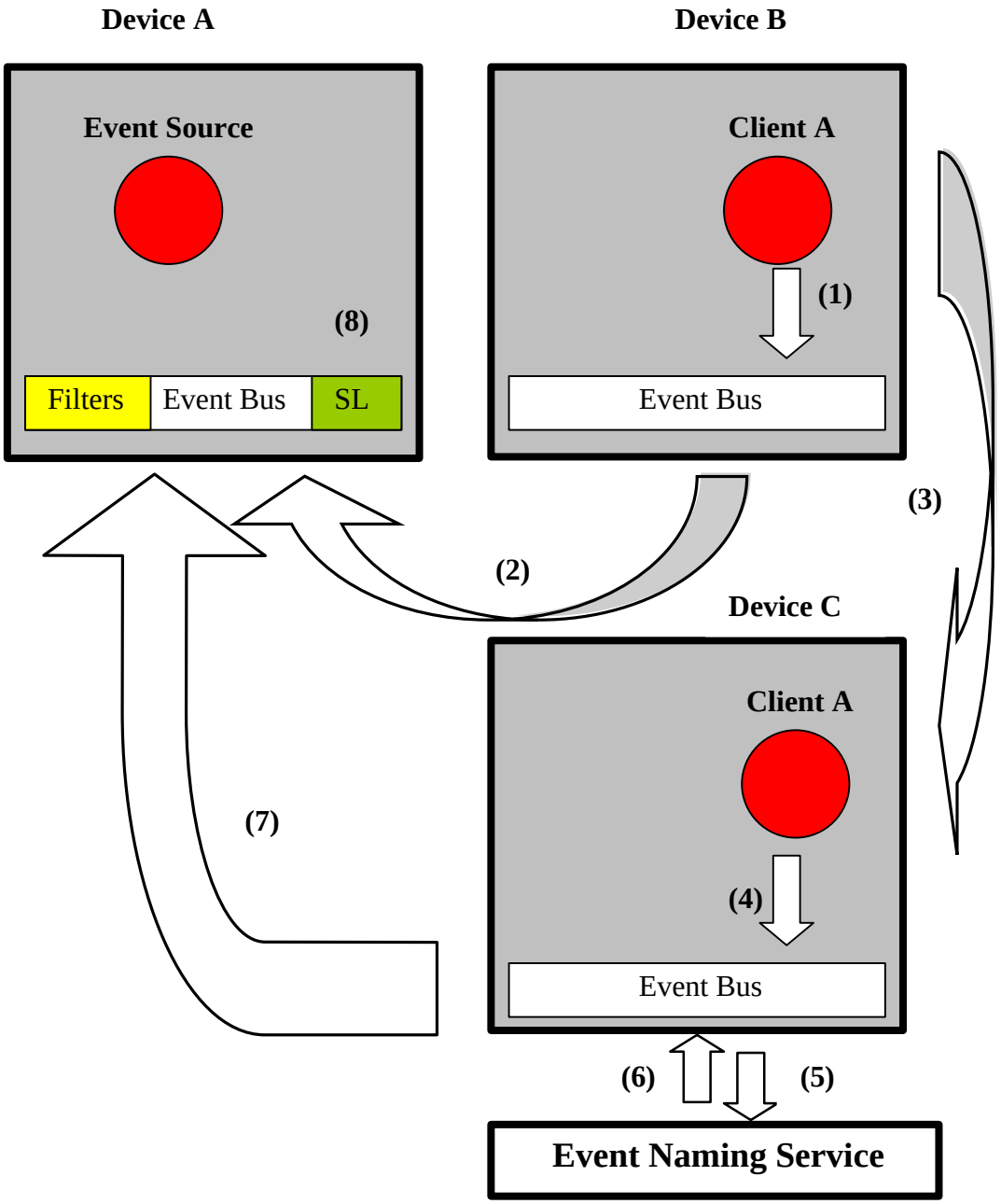


Figure 6 Mobility of Objects which Consume Events

The second case is the situation where the mobile object wishing to move location is a producer of events. In this case the sequence of step are as follows (see figure 7):

1. The mobile object, who in this case is a producer of events, notifies the local event bus of it wish to relocate.
2. The event bus then passes the filters and the subscription list to the destination of the mobile object. It also makes a request to the Event Naming Service to change the entry of the event to show the new location of the event source.
3. The destination event bus installs the filters and the subscription list. From then on the destination event bus handles the requests from mobile objects.
4. At this point the mobile object can transfer to the a new devise
5. When the mobile object has relocated it notifies the event bus of its readiness to produce new events.

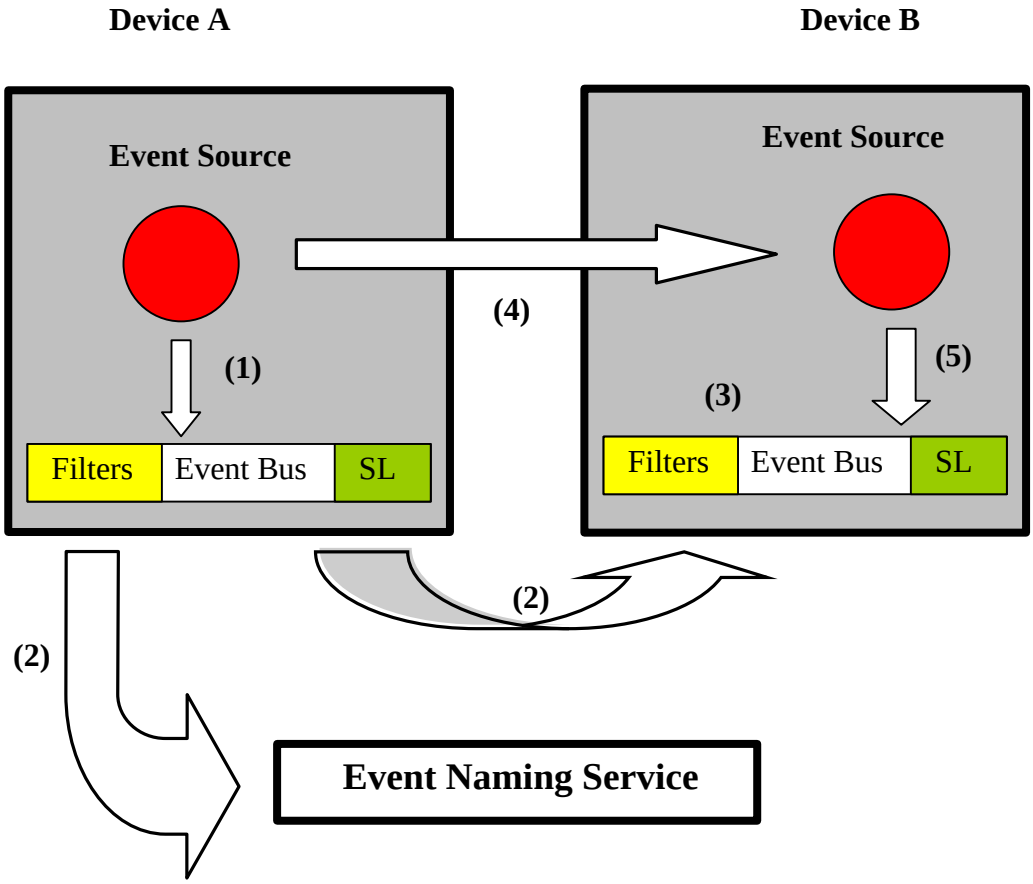


Figure 7 Mobility of Objects which Produce Events

3.11 Structure of the Event Buses

There are many different topologies that could be used in the structuring of the event buses to obtain more scalability. [6] describes several topologies that could be used; hierarchical, acyclic peer-to-peer, generic peer-to-peer and hybrid topologies. [6] also defines the pro and cons of each approach. At this stage it's not clear which topologies would best suit the event architecture described in this document.

3.12 Event Naming Service

The Event Naming Service is an integral part of this event service. It specifies the location of sources for each specific event at all times. It plays major role in

supporting mobility of objects within smart building environment. The Event Naming Service duty is to provide a tracking mechanism for mobile objects, which are acting as event sources. Also to provide a lookup facility for event buses in locating these mobile objects. All the information held on the mobile objects is held on a permanent storage device.

4 Summary

The architecture discuss in this document describes a general event service with the ability to support mobility of the objects. But to increase scalability of the service a better way needs to be found to structure the event buses to make them more efficient in the transferring notifications through the environment.

The work provides the foundation for future work on reconfigurable services as communication, via events, is now supported in an environment were objects are mobile.

5 References

[1] T.Walsh, P.A Nixon, S.A Dobson, “A Managed Architecture Mobile Distributed Applications”, TCD-CS-1999-03, http://www.cs.tcd.ie/publications/tech-reports/tr-index.99.html

[2] G.Cugola, E. Di Nitto, A. Fuggetta, "Exploiting an Event-based Infrastructure to Develop Complex Distributed Systems", In the *Proceedings of the 20th International Conference on Software Engineering (ICSE 98)*, Kyoto, Japan, Apr. 1998.

[3] G.Cugola, E. Di Nitto, A. Fuggetta, "The JEDI event-based infrastructure and its application to the development of the OPSS WFMS". Technical report, CEFRIEL, Milano, Italy, Sept. 1998.

[4] Jean Bacon, John Bates, Richard Hayton, and Ken Moody, “Using Events to Build Distributed Applications”, In the *Proceedings of the 1995 Secoud International Workshop on Services in Distributed and Networked Environments* (SDNE95). University of Cambridge Computer Laboratory, 1995.

[5] Object Management Group, Notification service, request of proposal, December 1996.

[6] Antonio Caraniga, "Architecture for an Event Notication Service Scalable to Wide-area Networks", PhD Thesis Politecnico Di Milano

[7] Chaoying Ma and Jean Bacon, “COBEA: A Corba-Based Event Architecture”, In the *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems*, Santa Fe, New Mexico, April 1998

Dynamic reconfiguration of FPGA nodes in a distributed computing system

Deliverable 1: Problem statement and planning

Dr.P.A. Nixon and Dr. S. A. Dobson

Motivation

In the mid-term, Information Management Assurance and Distribution calls for self-aware, reconfigurable, distributed computing environments of several hundred nodes that will provide an increased level of uninterruptible information services. A common real-time/non-real-time heterogeneous global information system will reduce time delays such as those associated with the movement of data from sensor to shooter.

In this effort a specialised portion of the above heterogeneous information system will be explored: Field Programmable Gate Array (FPGA)-based nodes. Algorithms that previously could only be seriously considered for software implementations can now be mapped to hardware in an FPGA so that performance and fault tolerance increases dramatically. Fault-tolerant designs need not be rigid, but can now be flexible without sacrificing throughput. This facilitates the development of adaptive systems that will begin to switch the emphasis from reaction to prevention of faults. Seamless communications within the global information system requires automatic accommodation of a range of protocols. With FPGAs, protocol designs can migrate from comparatively slow software to direct hardware implementations. At system runtime, migration of the FPGA's program code from node to node will permit changing the high-speed, hardware-based protocols seamlessly.

In light of the potential payoffs of uninterruptible information services, exploiting FPGA-Based nodes within an information grid needs to be investigated.

Project vision

Our vision is that the complete functionality of an FPGA node is exposed, possibly mimicked, and accessed through a CORBA-compliant object request broker (ORB) . We further see the system as being completely Java-based, both in terms of implementation and in terms of only running applications built on top of the Java virtual machine. This has a number of implications, including:

- Such an ORB is naturally *component-based*, in the sense that sub-systems can be installed very simply, allowing a variety of different algorithms and qualities of service to be supported.
- The use of components extends naturally up through applications, allowing investigation of systems built entirely from components and glue logic.
- Java’s type-safety can be leveraged to simplify a number of aspects of the reconfiguration process, notably in eliminating the need for low-level memory protection, process management and security checking.

- CORBA's distribution transparency can be leveraged to simplify programming of both high-performance and distributed applications, whilst retaining enough high-level information to optimise performance.

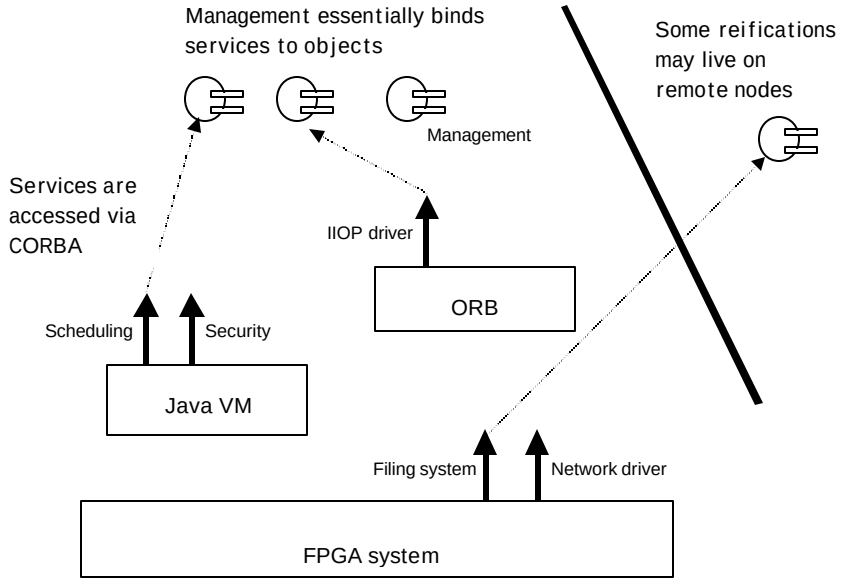


Figure 1: Proposed architecture

Initial planning and progress.

It should be noted that Dr. Proudfoot has left the University and FPGA expertise will now be provided by Dr. Doyle of Trinity College Dublin. Because of this change in staff the focus of the work will be on the distributed systems interface to FPGA systems and use in relocating FPGA computations. This re-focusing better reflects the skills of the remaing workers while still remaing true to the original proposal.

5 month schedule of activities.

October 1999

Elaboration of architecture design in document draft 1.
Proposed visit by Dr. Kwiat from Rome Laboratories.
Pricing and purchase of equipment as per proposal

November 1999

Implementation of Interface to FPGA gateway via CORBA

December 1999

Design and implementation of test cases to exercise FPGA interface

January 2000

Investigation of Reconfiguration issues based on CORBA interface.

February 2000

Vists to Rome Laboratories and planning of stage 2 of the project.

Dynamic reconfiguration of FPGA nodes in a distributed computing system

Deliverable 2: A Software Architecture

Dr. P.A. Nixon, Dr. S. A. Dobson and Mr. T. Walsh

Dept. of Computer Science, Trinity College, Dublin 2, Ireland
Phone: +353 1-608-1543, Fax: +353 1-677-2204
Email: Rene.Meier@cs.tcd.ie

1 Introduction

In this report we elaborate an architecture for dynamic reconfiguration of FPGA computation. We assume the availability of a an ORB interface to send controls to FPGA server software (currently under construction) and demonstrate feasibility of the reconfiguration approach to be adopted.

Current solutions to decentralised applications alone will not service the needs of this type of highly dynamic application. What is required is an inherently mobile solution detached from the present client-server model and the associated problems of scalability and flexibility [1]. These solutions will exploit mobile code because of its capability to dynamically change the bindings between code fragments and the location where they are executed [2]. This enables a migrating application to dynamically rebind to a generically named resource on a destination host while having the option of maintaining links to previously visited hosts.

We therefore present a design strategy for managed distributed applications in which we clarify the relationship between algorithm and location management. The motivation for this separation is the same as for the implementers of client/server systems such as CORBA, namely to implement network operations transparently. In this case the task is to implement the mobility aspect in a transparent manner thus allowing a single algorithm to work within a range of mobility policies. We derive an architecture for applications using this design strategy.

Section two discusses the characteristics of mobile distributed. Section three presents the design strategy, which is elaborated into an open architecture in section four. From section five onwards we explore how to manage fault tolerance using the architecture.

2 Mobile Distributed Applications

Code that executes on the majority of machines today may be classed as static code because it remains on a single node for the duration of its life. In terms of execution

speed it is the most efficient type of code, but it is not designed to ease communication with other nodes. In response to this observation, designers created the client-server paradigm [13] to allow executing processes on one node to avail of services on another, typically more powerful, machine. This concept was taken a stage further with peer to peer networking which enabled any machine to temporarily act as a server while communicating with client requests [6].

A logical advance of this concept was to free the actual processes from the confines of their nodes and allow the code to move to different nodes rather than just having remote method invocation. This approach is seen primarily in two distinct domains: intelligent agents [5] and object migration [14]. Object migration is taken to be the action of capturing the state of the executing object and sending it and it's respective class to a new node. The executing code will have been designed in such a way as to have little or no run-time state when migration occurs. Upon arrival at this new location the object will be restarted by the host system using a pre-migration determined method name. Object migration in this system does not involve thread migration. Thread migration involves serialising all run-time state state (i.e. local variables, intermediate results of computations that are on the stack, program counter, etc.) and continuing execution somewhere else. This type of migration is classified as strong mobility [1] but is not required in this instance. Strong mobility is on benefit to the application programmer who need not structure the code in any particular way to allow migration. Instead the system decides (using policy management) to migrate when the object has finished it's present task, similar to a type of checkpoint. This subject is discussed in more detail in section 5.1 The concept of moving computational 'know-how' in the form of code across nodes gives some distinct advantages over more static techniques. The advantages of migration include [3];

- Load sharing – By moving objects around the system, one can take advantage of underused processors

- Communication performance – active objects which interact intensively can be moved to the same node to reduce the communication cost for the duration of their interaction

- Availability – Objects can be moved to different nodes to improve the service and provide better failure coverage.

- Reconfiguration – Migrating objects permit continued service during upgrade or node failure.

- Location Independence – An object visiting a node can rebind to generic services without needing to specifically locate them.

The key concept in this type of system is its dynamic adaptability; the capability to conform to different situations at runtime. Systems such as these, despite having a simple concept, present a major challenge in actual implementation. A range of difficult dilemmas manifest themselves including servicing re-binding, administrating the binding type and maintaining the system.

Prior to migration an object can have bindings to many other different services and resources. Examples include printers, displays and databases. After migration some bindings will remain the same while others may require rebinding to similar typed resources on the new node. Thus to perform tasks and integrate successfully with applications on a given system objects must bind to resources at particular nodes and this dynamic re-binding must be automatic and transparent.

2.1 Binding

Foundational to the operation of mobility in distributed applications is binding. In [1] binding is described as the act of attaching to a resource by means of an appropriate type of reference. Such bindings are characterised by the fact they are transferable or not transferable, and by the strength of the binding (reference).

A binding by 'identifier' is the strongest type. In this case the resource is unique and cannot be replicated. Such bindings remain throughout an object's lifetime therefore any migration by the object will retain a network reference to the original resource unless the resource in question is transferable. In this case the resource could also be moved but bindings from other objects would need to be subsequently updated.

A binding by 'value' is weaker than by identifier. Such bindings declare that, at a given moment, the resource must be compliant with a given type and its value cannot change as a consequence of migration. This kind of binding is usually exploited when an object is interested in the contents of a resource and wants to be able to access them locally. In this instance the resource could be moved with the object if it is a transferable resource or bound by network reference if the resource is fixed.

A binding by 'type' is the weakest or the types. Such resources are generic resources that are typically available at any node. Examples include printers and displays. This type of binding is re-bound to the local resource on the new node. The management of such resources will be subject to the rules of the architecture. Binding types can increase in strength but never decrease. For example, if an object changes a database resource then that database has now gone from a duplicated resource to a unique resource and so the binding would need to be changed to an identifier type. This represents the resource's new uniqueness.

These binding types are similar to the methodology implemented in the FarGo system [16]. FarGo has five types of binding types, which it refers to as "complet references" because of the nature of the programming model. A mobility architecture [16] which requires mobility support to be built into each and every object is considered to be a fine-grained approach. Distributed Oz [19] implements a special programming language for this approach. The coarse grained approach [16], in contrast, divides the application into a set of processes, each of which is assigned its own distinct address space, and can move between hosts during its execution. Examples of systems taking this approach include Telescript [20] and AgentTCL [21]. An spplication designer must partition the application into a number of processes, and populate each process with a (possibly large)

number of objects. FarGo introduces a mid-grained approach in relation to the minimal relocatable entity (labelled a complet). A minimal relocatable entity is the smallest commodity which can be migrated. In FarGo it is typically a collection of objects which group together to form a particular task.

Separating the algorithm from its location will require a system that can provide resources transparently using a combination of services such as naming, trading and relationship [4]. At one level of abstraction each of these services will simply be managing binding. The benefit of abstracting a number of behaviours and managing them by re-binding, is that the algorithm can retain consistent access to a resource whose reference is managed by some external services. So the need for the programmer to explicitly cater for rebinding is now devolved to an external service. It is worth noting that the service can apply policies to the binding process to enforce a user or enterprise requirement such as security or performance policies. As an example, the maintenance of a distributed system involves all the usual tasks of incorporating performance tuning, system upgrades and fault tolerance. Since the system is not centralised it should be possible for one node to be taken down to allow upgrades to be performed without compromising the remainder of the system. Most, if not all, of these changes can be addressed by the introduction of our managed architecture. This particularly pays off in large-scale Internet systems where, because of their peer-to-peer nature, control and maintenance can become problematic.

This report presents a system that provides a managed and controllable view of a system's state space. By storing such information in a database structure multiple services (e.g., naming, trading, event services) can access the core of an application's state. The problem of distributed objects which can be disconnected and reconnected to different nodes is being solved, at present, using ad-hoc methods. A balance is required between full autonomy (ie agents) and controbility. The primary goal is to control object migration, not constrain it.

3 Design Strategy

In this section we explore some of the issues in designing information systems for the virtual enterprise. In particular we elaborate on the evolutionary nature of such an enterprise demonstrating the need for a managed solution. In parallel with this we explore the design strategies which must be employed and identify policy issues which must be captured during design and enforced in the implementation.

3.1 Separating form and function

Distributed object systems such as CORBA, DCOM and RMI present an abstraction in which objects may interact regardless of their locations. This allows the system designer to locate objects according to whatever high-level criteria are appropriate without complicating the application's code. However the locations are fixed at object creation -

once located, an object does not migrate - which is a considerable handicap, preventing a long-lived system from adapting to changing conditions by re-locating services.

Operating systems research in this area has concentrated on process migration, moving the complete execution context of a running process from one machine to another. The great advantage of this technique is that the process is suspended during migration, completely masking the change of location. This advantage is also a critical disadvantage for systems in which location is a factor in acquiring
resources, since the process will be unaware that its previous bindings have been rendered obsolete.

At the other extreme, several authors have suggested agent communities as an approach to virtual enterprises. Agents encapsulate algorithmic functionality with control of their own mobility, allowing them to wander around a system performing work locally wherever possible (In some agent systems there is no remote communication, so all agent interaction occurs locally after migration to a common point.). Although such decentralised control improves communication efficiency, local decision-making is known to have severe drawbacks in complex systems. The combination of locally optimal solutions is often highly sub-optimal, and the interactions of several different control policies (by agents pursuing different goals) rapidly becomes impossible to analyse or predict.

A related problem is the coupling of the function an agent performs and the location at which it performs that function. It can be argued that this coupling stems from the object-oriented principle of complete encapsulation, allowing an agent to be treated as a "black box". However, the alternative view is that this coupling reduces the ability to re-use the agent in varying circumstances, since it may be difficult to change its movement logic while re-using its functionality.

While agent systems undoubtedly have a place in the virtual enterprise (and especially as information gatherers outside the system), we feel that they cannot as yet be recommended for mission-critical infrastructural tasks. We believe that the best approach for virtual enterprise systems is to separate the algorithmic logic of a component from the logic used to locate and re-locate components - essentially divorcing the function of an application from its form over the network. This approach is already used tacitly by components assuming a fixed location: it may be generalised to include components which allow themselves to be migrated by an external entity.

The separate configuration logic can be handled in a number of ways, for example by providing plug-in location managers for components. This would still suffer the problems inherent in local decision-making. A completely centralised policy manager is superficially attractive but would become a severe bottleneck for large and/or highly dynamic applications. A combination approach is to use a location management component which is itself completely stateless (and which may therefore be replicated freely) but which references other information sources in making its decisions. By pushing the problem back, we gain the ability to use centralised, federated or fully distributed information sources without affecting the design or interactions of components.

The use of a separate configuration component raises a number of other design questions. Chief among these is the way in which re-configuration is seen by components. Can a component request to be moved to a specified location (the agent approach)? Can a component be forced to move, and what are the implications of this on resource binding and concurrency control? Can a component prevent migration, and what does this imply for system predictability? Different scenarios will give different answers to these questions, all or any of which might be desirable. By divorcing application from policy decisions in our architecture we introduce the flexibility to choose the most desirable behaviour for the system at any given time.

3.2 Stability in the face of change

The "software crisis" has shown that current software engineering practices are often less than adequate at handling relative fixed systems. Dynamically adaptive systems therefore offer a severe challenge to the design of systems, programming languages and components. The essential problem is to provide the system designer, system manager and component implementor with a platform that is sufficiently flexible to capture the range of re-configuration operations while retaining sufficient stability to be usable in practice.

We may observe that most systems' architectures change considerably more slowly than their objects. That is to say, a role in a system is longer-lived than the particular object that fulfils it. Thus the distributed object approach of interacting with objects whose names are explicitly known is problematic in the face of long-lived applications.

System architecture is usually considered as a precursor to design rather than a run-time concept. If we maintain a machine-readable description of an application's roles and relationships, we may make high-level role information available directly to components. This would allow, for example, a component to construct a reference to "the user database" as an architectural concept, regardless (at least in some senses) of the database object providing that service.

While the most obvious application of this approach is in providing high-level binding, it may also be applied to location-sensitive resource acquisition if the architecture provides a relationship between a resource and its physical location (which may change across time). This allows a component to specify a binding to a local resource - "the local printer", for example - and have that binding remain valid in the face of changes in location. Thus the same technique can be used to provide both stability in the face of change and sensitivity to local conditions. Several issues arise immediately from this approach.

The first concerns role changes, which must be propagated to all components making use of that role. One may adopt an eager approach (sending notifications to all affected components) or a more lazy approach (forcing component to re-acquire the correct object before use). Both approaches are pathological under some circumstances.

A second issue involves the effects of migration, which may invalidate a specification even if the roles themselves are unaffected. This requires some mechanism in the underlying run-time system to detect location changes, even if location changes are masked from the component in other ways.

The third issue concerns the way in which a component interacts with a resource specified by role. In most current systems the resource will still at some level be represented by explicitly known references, which will be needed for any interaction to occur. We must therefore address the manner in which these names are unpacked from their role-based specification. We must also address the concurrency control issues implicit in changing the object fulfilling a role while the reference is unpacked.

4 A Managed Architecture for Mobile Distributed Applications

The structure of a generic migratory distributed application is displayed in figure 4.1. It shows how each component (also referred to as an autonomous object) can exist on a different node and yet remain in communication with other components of the application. Here the term node refers to a single processor, typically a single host. The circles represent components, while the dashed lines represent the various network references, or simply links, between them. The components are capable of being migrated to another node yet still carry on its computation, and more importantly its role in the computation. This is the goal for a real world system but prior to building such a system it is necessary to describe a disciplined abstract structure.

The complete structure of the architecture can be encapsuated in a logical database. It represents a rational method to store such a configuration because it is well structured, secure and retains data integrity automatically. The logical database can be implemented as a single, distributed or federated database, and we use the central database term solely to convey the methodology to be used in implementing such a system.

Different interfaces to the database allow access to the data. These interfaces are in the form of the standard services such as those specified by the OMG, RM-ODP, ODMG, and IETF. Due to the fact that they are stateless they represent logical filters to all data. For consistency we assume the definitions of the OMG for discussions of services named below.

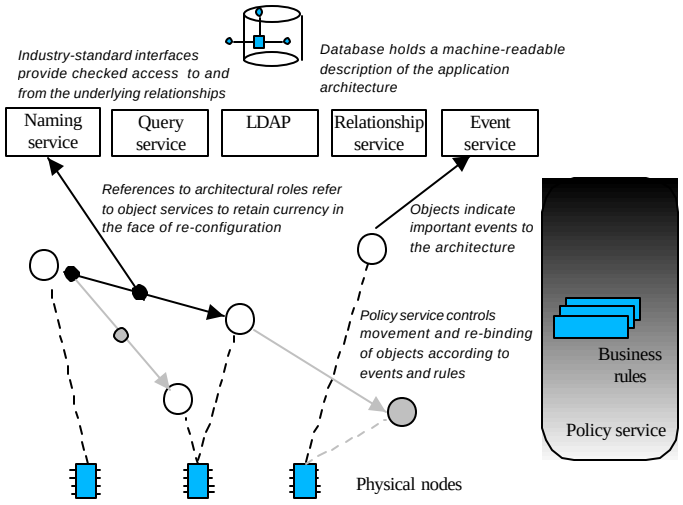


Figure 1: A Managed Architecture Mobile Distributed Applications

The relationship service and naming service will assist migrating components to rebind to well specified or generic services. For example, should a migrating component wish to avail of a standard printing service on a destination host then the relationship service will provide the appropriate resource name. If the docking component has special 'secure' privileges then the relationship service may specify a secure output device such as the manager's printer while 'normal' components are referred to a standard office printer. Using this evaluated resource name a concrete binding to the particular resource is returned from the naming service. The Event Service [4] allows objects to communicate using decoupled event-based semantics. The Query Service provides an interface in much the same way as SQL provides an interface to databases.

All the services see the same data in the database. Their purpose is to give the component a view of just the data needed to provide the bindings between resource names and their object references. This provides a seperation of data from the interfaces used to access it. Migrating components use the service interfaces to reconfigure their resource bindings. Each migration is different and can have different consequences. Use of standards allow the architecture to evolve as more interfaces become available over time and therefore, such an open system has the potential to do for migratory applications what OMG's CORBA did for remote object invocation.

Application component failures in large scale distributed systems [1][9] are inevitable. *Fault tolerance problems* [7], associated with the use of large scale distributed systems, arise because application components may eventually fail. These failures are caused by hardware problems, operator mistakes or software faults [3]. Within most environments, and in particular within an embedded environment, such failures are not acceptable. In our problem domain we are interested in fault tolerant service provision. By fault tolerant service we mean:

A service in a distributed system is called fault tolerant when it behaves according to its specification even in the presents of failures in parts (processor, media, communication link, other service) of the distributed system on which it depends [9].
The author [8] identifies four typical causes of application component failures in most environment. All of them can be overcome by providing a fault tolerance mechanism that re-connects a client from a failed service to a similar backup service. This report introduces such a mechanism, which is described more completely in [8].

5 Fault Tolerant Application Components

We now apply this generic architecture to build a reconfigurable and fault tolerant system. The system is based on a three-tier client-server architecture. It is implemented using CORBA [2][6][10], a distributed object middleware specified by the OMG [10]. It includes several databases, which are kept consistent using a data replication protocol. Servers and their services (service objects) are managed by a so-called Service Manager, which, together with the appropriate hardware, guarantees them to be *fail silent*. We assume the existence of low level fault tolerance mechanisms. We introduce a mechanism that re-connects clients from unavailable master services to backup services. Master services may not be available because of failures or for maintenance reasons. Currently, clients connected to an unavailable master service are not able to retrieve data requested by a user, despite the presence of other similar services that could provide the requested data. The basic assumption is that re-connection delays and even a lower access performance are acceptable, but a total loss of a service is unacceptable.

5.1 Some Requirements of Fault Tolerant Application Components

We propose an architecture to support the development of fault tolerant distributed application components. Central to the architecture is the use of the OMG Object Trading Service [10] as the mechanism to advertise and manage service offers of fault tolerant application components.
Fault tolerant issues must be *hidden* from the client application program and therefore from the user and be Object Request Broker (ORB) [10] *independent*. The suggested architecture must support *on-line* application component management by configuration adjustment, without re-booting the rest of the system. It should provide a highly available system with a good trade off between *system scalability* and *service performance* without resorting to purchasing expensive fault tolerant hardware.

6 Object Trading

In this Section, we first introduce OMG's CORBA Object Trading Service [10], which is included in the suggested architecture as the mechanism to advertise and manage service offers for fault tolerant application components.

6.1 OMG Object Trading Service

There are several possible approaches for service consumers (clients) to retrieve a reference of a service (service object) provided by a server, that may be located somewhere in a distributed system. Such a reference can be available on the client side, e.g. looked up in a table or read from a file or, in a more dynamic approach, can be retrieved from a naming service such as the OMG CORBA Naming Service [10]. In these approaches, in order to retrieve a service object reference, clients need to know the exact name of the desired service object a priori.

The OMG Object Trading Service lets clients dynamically discover service objects based on the type of service they provide. It is like yellow pages for service objects in a distributed system. A server advertises its service objects with a Trading Service. Clients use the Trading Service to discover service objects that match their needs **Error! Reference source not found.**. This is shown in figure 2 and achieved as follows:

1. A server registers (*exports*) its service objects with the Trading Service. By doing so, a server gives all the relevant information about its service objects to the Trading Service. This service offer includes the service object reference, the service object name and the service object properties. Clients use the reference to connect the service object and to invoke on its operation. The name includes the operations to which the service object will respond and their parameters and result types. The properties are name value pairs, which describe the capability of the service object. The Trading Service maintains all the service object information in a repository.
2. A client makes a request (*imports*) for a specific service object type. Based on the properties of the service objects, the Trading Service performs a matching algorithm to return the result to the client.
3. Based on the Trading Services information, it is now the client's responsibility to decide whether or not to *invoke* a service object on a server.

A Trading Service may use an inter-trader protocol to extend its search to other Trading Services. Such a schema is called *federated trading*. We do not consider enhanced trading schemas such as federation in this report, although the architecture can be extended to use federation easily.

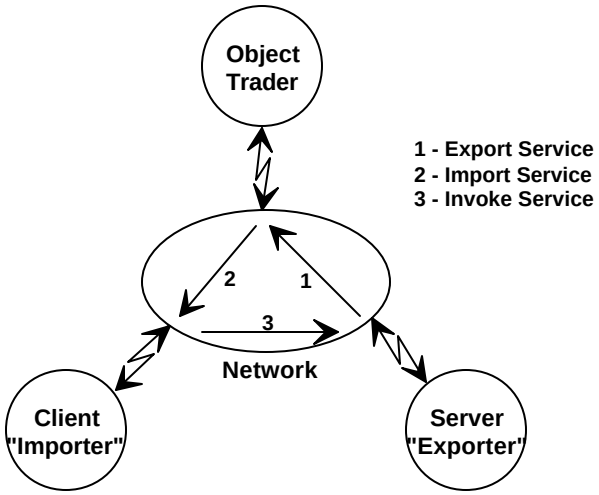


Figure. 2. The OMG Object Trading Service.

Another reason for using an object Trading Service is system stability and scalability. Distributed systems consist of a large number of bindings between clients and servers. Since these bindings are not reliable, re-discovering and re-connecting at run time helps in overall system scalability and stability since the bindings can always be reorganised.

6.2 The Environment

The proposed architecture was designed to manage fault tolerant application components of a FPGA environment but can easily be applied for any client-server based architecture.

6.2.1 Environment Terminology

The environment consists of:

- Client: A client is a program, operated by a user, for which a server performs some computation.
- Server: A server is a physical machine with several processors.
- Service: A service (service object) is a software application component running on a server.
- Location: A location is a collection of services running across one or more servers, providing functionality to a community of clients (users).

6.2.2 System assumptions

The target system is based on a three-tier service based architecture. CORBA is used for the implementation of the communication link between the client-tier and the middle-tier (service-tier).
Clients (users) are grouped together in so-called locations accessing service objects in the middle-tier of the system, that may be located on one or more servers. Service objects are logically associated with locations. A server may provide service objects for one or more locations. The servers access a distributed database (database-tier) that is kept consistent using a data replication protocol. The client-tier and the middle-tier are of particular concern. Within the middle-tier, the features of the Service Manager and the Notification Service will be used in the design.

- The *Service Manager* maintains the status of each of the service objects, using pings, a request that asks if the service is still running. It will shutdown and / or try to restart service objects that failed. The status of each of the service objects will be used by the Notification Service to publish service object status notifications. The Service Manager, supported by the appropriate hardware, guarantees the service objects to be fail silent.
- The *Notification Service* publishes several different types of notifications. A particular notification type is published in the event of a service object status change. To receive notifications, clients register with the notification type they are interested in.

7 Managing Fault Tolerance

To solve the fault tolerance problem identified in Section 1, an OMG Object Trading Service is introduced to the system. The included component must have appropriate

service types, including a sequence of property structures that describe services, to export and import service offers. Property structures include client group identifier lists. Clients with similar needs and location are grouped together and are given a unique group identifier. Servers offer their services to a particular client group by including client group identifiers in a service offer. Clients then query the Trading Service for service offers which include their group identifier.
The basic architecture is a *simple low cost solution* that solves the fault tolerance problem without involving the Notification Service. Clients query the Trading Service for master and backup service offers and cache the retrieved service references. If the service in use fails, the service user (client) is re-connected to the backup service. During re-connection, the service user is idle. Then, the client starts pinging the original service and re-connects the service user to it as soon as it is back on-line. Pinging is inefficient and causes unnecessary network traffic. To eliminate this, an improvement is proposed, which makes use of the Notification Service. Within the improvement, clients use notifications to detect service failures and therefore need not to ping services anymore. This results in less network traffic and reduced service user idle time. Notifications are also used to automate service offer maintenance and therefore further minimise network traffic. In addition, the trading service's dynamic properties are used to provide load balancing.

7.1 Service Offers

Servers cannot export service offers unless the Trading Service has appropriate service types. The Trading Service can contain a number of such service types that describe services. Service types consist of a type name, an IDL interface ID and a sequence of property structures.
Service offers describe a specific service provided by some server, based on the information defined in a service type. Service offers consist of a reference that is the actual service object reference and a sequence of properties, where each property is a name-value pair.
Each service property of a property structure is qualified by mode attributes. Mode attributes are read-only and mandatory; combinations of both are also allowed. A value of a read-only property can initially be set when its offer is exported to the Trading Service, but cannot be modified afterwards. A value for a mandatory property must always be provided when a service offer is exported to the Trading Service.
When the Trading Service processes a client's query for a service offer, it gathers a sequence of offers together by narrowing down the set of potential offers. The query input is used to determine whether or not an offer is of an appropriate service type. The property constraints are used to determine whether or not the offer matches. The Trading Service's matching algorithm is more fully described in [8][10].

7.1.1 Service Offer Property Sequence

Table 1 shows the core property sequence of the fault tolerant service offers and includes property value examples. In order to keep service offers compatible, the mode attribute of additionally appended properties must not be mandatory.

Property name	Data type	Mode attributes		Value example
		Mandatory	Read-only	
ServiceTypeName	String	✔	✔	ft_echoService_if
ServerName	String	✔	✔	EchoServer1
ServiceName	String	✔	✔	Echo Service One
MasterList	String	✔		"0001-0002"
PrimaryBackupList	String	✔		"0003-0004"
SecondaryBackupList	String	✔		"0005-0006-0007"
OfferIsValid	Boolean	✔		True
ServerUtilization	Long			18 %
NumOfUsersOnServer	Long			6

Tab. 1. Property Sequence.

7.1.2 Mode Attributes

The property sequence includes properties that have different mode attributes. ServiceTypeName, ServerName and ServiceName's mode attributes are mandatory and read-only. These properties uniquely identify a service. A property value must always be provided and cannot be changed during service lifetime. If one of these properties must be changed, the original service offer must be withdrawn and then be replaced by the new service offer. The mode attribute of MasterList, PrimaryBackupList, SecondaryBackupList and OfferIsValid is mandatory. These properties describe the target client groups and the status of the offered service. These values may change during service lifetime, e.g. a service may become backup for another client group. ServerUtilization and NumOfUsersOnServer's mode attribute is normal (neither mandatory nor read-only). The feature that supplies the values for these properties may not be supported by some services. Such services ignore these properties simply by not providing a value for them.

7.1.3 Names and Values

ServiceTypeName, ServerName and ServiceName contain string values, which uniquely identify a service. A particular service type (ServiceTypeName) can be provided by several servers (ServerName), which can supply several similar service instances (ServiceName). These values are used when clients register with notification channels they are interested in and might also be displayed to the service user.
MasterList, PrimaryBackupList and SecondaryBackupList contain string values that include a list of client group identifiers. Client group identifiers must be unique and need to be separated within the string by a delimiter. A client queries the Trading Service for an offer that includes its group identifier in the MasterList or in one of the BackupLists respectively. The OfferIsValid boolean value marks a service offer as valid or invalid. A service offer is marked as invalid when it is temporary out of service, e.g. for

maintenance reasons. Thus, clients expect such services to be back on-line eventually. This is not expected if no service offer was found in a particular list, e.g. there might be no secondary backup service available for a particular client group.
ServerUtilization and NumOfUsersOnServer's long values, which have to be implemented as dynamic properties, may be used to select the best available offer in terms of load balancing. Either the client selects the service offer with the lowest server utilisation and/or number of users or, when configured appropriately, the Trading Service does by exporting only the best available service offer.
Using property names and values, clients query the Trading Service for a particular service type (ServiceTypeName) that is master (MasterList) or backup (PrimaryBackupList or SecondaryBackupList) for its group identifier. The imported service offer includes server and service name (ServerName, ServiceName), the current service status (OfferIsValid) and load balancing information (ServerUtilization, NumOfUsersOnServer).

7.2 Fault Tolerance Architecture

The following architecture introduces the OMG Object Trading Service to an existing system as the mechanism to advertise and manage service offers for fault tolerant application components. The *basic architecture* does not require notifications. The improvements may be build on top of it to increase re-connection performance, reduce maintenance and to include additional features such as load balancing. These improvements require notifications generated by the system's Notification Service. Even in the presence of a well designed, reliable and highly efficient notification service, there is always a possibility that *clients detect service failures **before** receiving the corresponding notification*. Clients are enabled to handle this by first implementing the basic architecture and by then providing extensions.
Table 2 shows the tasks that have to be performed to provide fault tolerance within the system. The following Sections refer to table 2 when discussing the tasks.

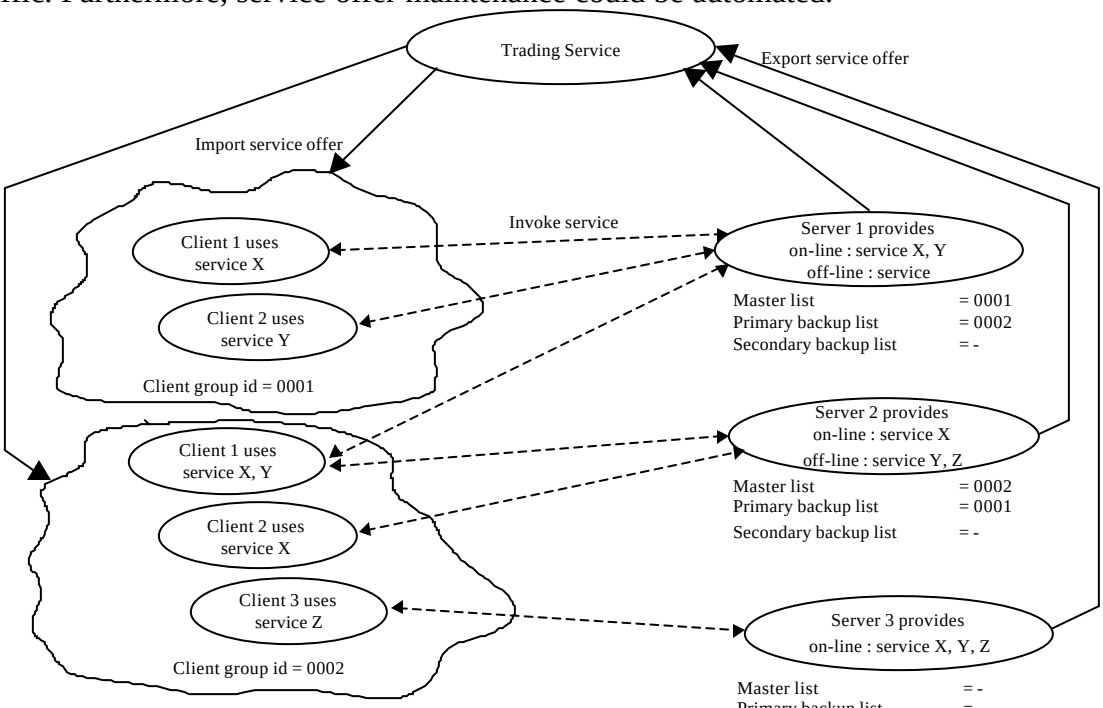
Where	What
Clients	[A] have to detect a service failure [B] need to obtain a backup service reference [C] have to re-connect to the backup service [D] have to re-connect to the original service as soon as it is back on-line
Services	[E] have to be maintained in order to be fail silent
Servers	[F] have to be maintained in order to be fail silent [G] have to provide values for dynamic properties
Trading Service	[H] service offers have to be exported to it [I] temporary invalid service offers have to be marked [J] service offers that are no longer valid have to be withdrawn [K] dynamic properties have to be supported for load balancing

Tab. 2. Tasks to Provide Fault Tolerance.

Due to the space limit, this report will not discuss load balancing issues [G], [K]. A more complete description may be found in [8].

7.2.1 Architecture Design

Figure 3 shows the basic solution that does not involve the Notification Service. Clients detect service failure [A] by the time they invoke on a service object, even if the service failed earlier. An invocation on a failed service will eventually return an exception. The service user is idle during failure detection time, which depends on timeouts and network topology. Backup service references [B] were pre-fetched and cached and should therefore be available. Thus, re-connection to the backup service [C] will be efficient. Because of the lack of a cache updating mechanism, backup service references might have become invalid. After detecting an invalid service, clients start pinging the original service and re-connect [D] to it as soon as it is back on-line.
Services and servers are maintained [E][F] by the Service Manager. The maintenance of service offers, such as exporting [H], marking invalid [I] and withdrawing [J], is not automated and has to be performed by IT personnel.
Although this solution suffices, several improvements may be made. Service user idle time can be reduced and the cached service references should be maintained. Pinging the original service before re-connecting to it is not efficient and causes unnecessary network traffic. Furthermore, service offer maintenance could be automated.



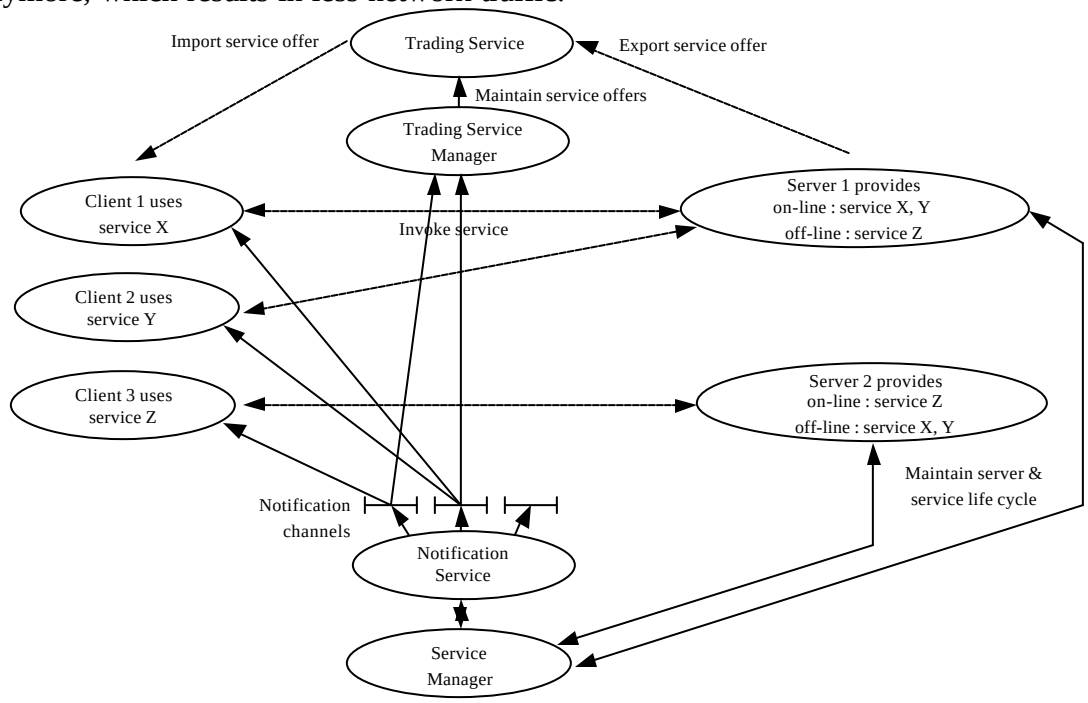
Legend:

- Client 1 (group id = 0001) invokes master service X on Server 1
- Client 2 (group id = 0001) invokes master service Y on Server 1
- Client 1 (group id = 0002) invokes master service X on Server 2
- Client 1 (group id = 0002) invokes primary backup service Y on Server 1
- Client 2 (group id = 0002) invokes master service X on Server 2
- Client 3 (group id = 0002) invokes secondary backup service Z on Server 3

Figure 3. Basic Architecture.

Figure 4 shows an improvement on the basic architecture that includes the Notification Service and introduces another component, called Trading Service Manager, into the fault tolerance mechanism.

Clients receive notifications whenever a service goes out of service and when it is back on-line. Invalid service references are detected [A] and clients are re-connected [C] to cached [B] backup service references in most cases without service user idle time. There is no need for pinging the original service. Clients re-connect [D] to the original service reference immediately after receiving the corresponding notification.
Services and servers are maintained [E][F] by the Service Manager. The Trading Service Manager also receives notifications. It is responsible for updating Trading Service's service offers, that is to mark temporarily invalid service offers [I]. Thus, clients will recognise invalid service offers when receiving them from the Trading Service; this reduces the possibility of service user idle time further. Servers could export [H] their service offers at start-up time and withdraw [J] them before shutting down. Therefore, IT personnel has to maintain service offers of crashed services, that are not going on-line again, only.
The improvements on the basic architecture include significantly reduced service user idle time due to re-connection in background and improved client cache consistency, and automated service offer maintenance. There is also no need to ping the original service anymore, which results in less network traffic.



Legend:

- Client 1 and Client 2 receive a notification: "Server 1, Service Z out of service"
- Client 3 receives notifications : "Server 2, Service X out of service"
  "Server 2, Service Y out of service"
- Trading Service Manager receives notifications : "Server 1, Service Z out of service"
  "Server 2, Service X out of service"
  "Server 2, Service Y out of service"

Figure 4. Improved Architecture.

The drawback of the improvement is the introduction of another component, the Trading Service Manager, which in the absence of replication is another single point of failure.

Attention has to be paid to the delivery order of the notifications. When a ”service offer updated” notification is generated, the Trading Service must be updated before clients query for the affected service offer. This can be achieved by including a logical timestamp (sequence number) in the notification. This sequence number is added to the service offer by the Trading Service Manager when updating it. Clients are then able to verify whether they received an up to date service offer from the Trading Service. Another way to guarantee notification delivery order is to send them to the Trading Service Manager only, which updates the Trading Service and then forwards them, via the Notification Service, to the clients.
This report does not address component replication to avoid single point of failure, e.g. in the Trading Service Manager. The system has already addressed this issue for its components, e.g. the Notification Service. Thus, we expect this problem to be overcome in a similar way.

7.3 Implementation and Integration

A prototype of the basic fault tolerance architecture was implemented, as shown in figure 4, in the Java Programming Language [5] using Iona’s OrbixWeb and OrbixTrader [4]. Client side fault tolerance algorithms were implemented as smart proxy classes, in order to hide them from the client application program. Unfortunately, the smart proxy feature is ORB vendor specific. A way to implement a fault tolerance algorithm ORB independently is to place it between the IDL interface and the client application program, thus removing transparency.

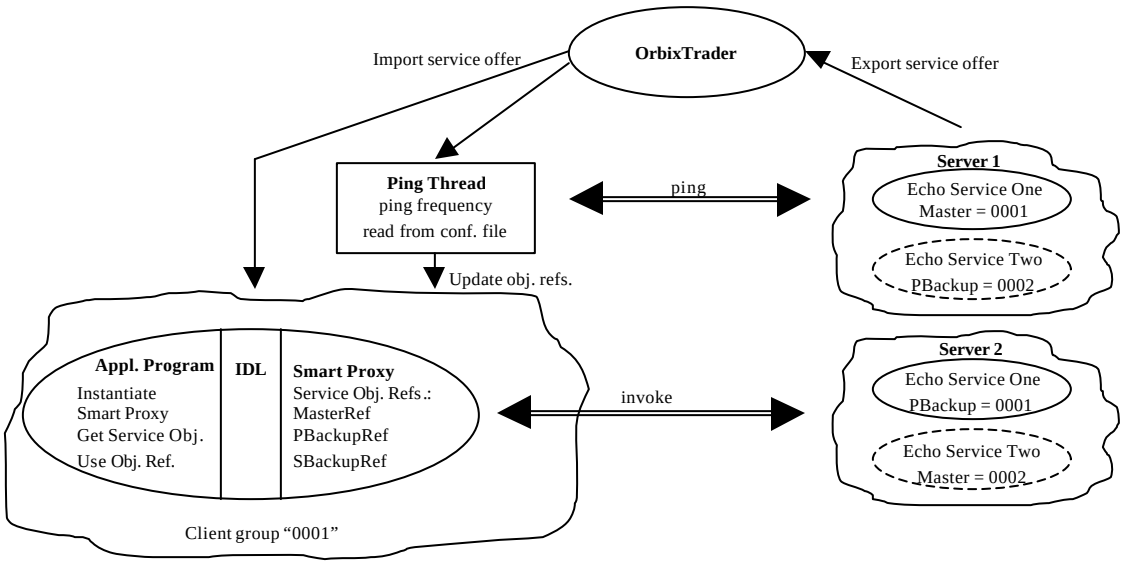


Fig. 4. Basic Architecture Implementation.

Both proposed architectures were designed to fit into the chosen system and to solve the fault tolerance problem identified in Section 1, by introducing new software components.
Fault tolerant application components are managed by configuration adjustment, e.g. an out-of-date service offer can be withdrawn and replaced on-line, and by client configuration files, containing a few parameters such as client group identifier, to support system scalability. Performance, in terms of service user idle time and network traffic overhead, increases significantly with the suggested improvement of the architecture. As

mentioned above, a trade off between ORB independence and fault tolerance hiding had to be made. Although several ORB's support a smart proxy feature, that is used to hide fault tolerance from client application programs and therefore from users, it cannot be used without loosing portability in terms of ORB independence.

8 Evaluation

The prototype implementation of the basic fault tolerance architecture was successfully tested on both, Sun Solaris and WindowsNT platforms. During integration testing, services where killed to force client smart proxies to re-connect to backup services and to re-connect to the original service, as soon they were re-started.
The performance of the implemented prototype is evaluated here. The duration of method invocation on a service object running on a remote server, as well as re-connection to backup and re-connection to original service object running on a remote server was measured. The method invocation duration times were taken for 1000 invocations on a simple method that returns an integer value. The results were averaged to get the duration of a single invocation. The server was connected first using bind and second using the fault tolerance mechanism.
Service re-connection duration was measured by taking the time of an interrupted service invocation. All measurements were made with OrbixDaemon, OrbixTrader [4], client and server process running on the same Sun Solaris Ultra SPARC box and OrbixWeb3.0's [4] default configuration.

Sample [ms]	1	2	3	4	5	6	7	8	9	10	Average
Bind	3.961	3.946	4.174	4.161	3.922	3.903	3.959	3.942	4.083	4.033	4.008
Fault tolerant	4.021	4.034	4.041	4.006	4.034	4.013	4.061	4.056	4.043	4.051	4.036

Tab. 3. Method Invocation Duration.

Sample [s]	1	2	3	4	5	6	7	8	9	10	Average
Re-connection to backup service	12.227	12.217	14.521	12.166	14.615	12.155	12.229	12.328	12.303	14.568	12.933
Re-connection to original service	1.785	1.695	1.946	1.686	1.695	1.766	1.675	1.957	1.780	1.696	1.768

Tab. 4. Service Re-connection Duration.

The method invocation measurements in table 3 show that invocation time on a bound service object and on a fault tolerant service object are *essentially identical*. The difference between the two is that the invocation on the fault tolerant service object uses the provided smart proxy, whereas the invocation on the bound service object uses the default proxy. These are (almost) identical in absence of a service failure.
The re-connection to backup service, shown in table 4, consists of *failure detection*, which uses OrbixWeb3.0's [4] COMM_FAILURE timeout, *message transmission*, which depends on the network topology and *client algorithms.* The re-connection to original service, also shown in table 4, consists of pinging and therefore starting up the original (failed) service, updating the client's cache, selecting the best available service object reference and killing the ping-thread.

Re-connection to backup service and re-connection to original service may both cause service user idle time. The measurements show that user idle time is within an acceptable range. The worst case is less than 15 seconds, opposed to minutes or even hours due to a temporary or total loss of a service in absence of a fault tolerance mechanism.

9 Conclusions

This report describes an architecture to transparently manage fault tolerance in a large scale distributed system. The presented architecture is designed to fit into an existing system and allows the development of fault tolerant application components. Of primary importance to our solution is the inclusion of the OMG CORBA Object Trading Service into the fault tolerance architecture as the mechanism to advertise and manage service offers for fault tolerant application components. The mechanism enables clients transparently to detect a failed connection to a service object, to discover a similar backup service object and to re-connect to it. The design of the architecture allows application component management by configuration adjustments, without re-booting the system, supports the different needs of the system's clients and adequately addresses the scalability requirements of the system's infrastructure.
A prototype of the suggested mechanism has been realised and evaluated. The implementation shows that performance issues are appropriately addressed and that performance can be improved by completely implementing the architecture. The limitation of the implementation is the trade off that had to be made between fault tolerance hiding and ORB independence. In order to hide fault tolerance algorithms from the client application program, an ORB specific feature, so called smart proxy, was used. This issue is subject of further research. In conclusion, it has been demonstrated that this architecture supports the development of fault tolerant distributed application components, which results in *improved availability.*

10 Next steps

The final stage of this study requires us to demonstrate the feasibility of controlling FPGA computation with this generic architecture. To achieve this we are undertaking a interface development for an FPGA system.

The results of this development and the final assessment, particularly with reference to moving FPGA computation between nodes will be elaborated in the final report.

11 References

[1] J. Bacon, *Concurrent Systems.* Addison-Wesley, 1993.

[2] S. Baker, W. Cahill and P. Nixon, *Bridging Boundaries - CORBA in Perspective.* IEEE Internet Computing, Volume 1, Number 5, September/November 1997.

[3] M. Banâtre and P. A. Lee, *Hardware and Software Architectures for Fault Tolerance*. Springer Verlag, 1994.

[4] Iona Technologies PLC, URL = http://www.iona.com.

[5] Java Programming Language, URL = http://www.java.sun.com.

[6] S. Landis and S. Maffeis, *Building Reliable Distributed Systems with CORBA*. Theory and Practice of Object Systems, John Wiley, New York, April 1997.
[7] P. A. Lee and T. Anderson, *Fault Tolerance: Principles and Practice (second edition).* Springer Verlag, 1990.
[8] R. Meier, *A Framework Providing Fault Tolerance Using the CORBA Trading Service.* MSc. Thesis, University of Dublin, Trinity College, September 1998.
[9] S. Mullender, *Distributed Systems.* Addison-Wesley, 1993.
[10] OMG, Object Management Group, URL = http://www.omg.org.
R. Orfali, D. Harkey and J. Edwards, *Instant CORBA*. John Wiley & Sons Inc.,